情 報 学 基 礎 16－3
（1989．12．7）

ワイルドカードを用いた2次元パターン照合

竹田 正幸

九州大学工学部電気工学科

パターン照合問題とは，テキスト中のパターンの出現をすべて求める問題である。本稿では，文字種（ピクチャ）を含むパターンを対象としたパターン照合問題を取り扱う．例えば，英小文字(a,b,...,z)を表すピクチャをA，数字(0,1,...,9)を表すピクチャをNとするとき，abNN，a7$NNNNA$などのパターンをテキスト中より検出することを考える．複数の文字列パターンに対する効率的な照合アルゴリズムの1つとしてAho-Corasick法(AC法)がある．この方法では，まず与えられた複数のパターンからパターン照合機械とよばれる一種の有限状態機械を構成する．これにより，テキストをただ一度走査する間に同時に複数のパターンを照合することができる．このアルゴリズムの自然な拡張として，ピクチャを含むパターンを同時に複数個探索する効率的なアルゴリズムを述べる．さらに，その応用として，ワイルドカードを用いて2次元パターン照合をより柔軟に行う方法について論じる．

Two-Dimensional Pattern Matching Using Wild Cards

Masayuki TAKEDA

Dep. of Electrical Engineering

Kyushu University, Fukuoka 812, JAPAN

The pattern matching problem is to find all occurrences of patterns in a text string. In this paper, we consider patterns with pictures. For example, let A be a picture for a, b, ..., z, and N for 0, 1, ..., 9. Then we consider patterns such as abNN, a7$NNNNA$, ..., etc. For multiple string patterns, the Aho-Corasick algorithm, which uses a finite state pattern matching machine, is widely known to be quite efficient. As a natural extension of this algorithm, we present an efficient matching algorithm for multiple patterns with pictures. Moreover, we discuss an application to the two-dimensional pattern matching.

1 Introduction

The pattern matching problem is to find all occurrences of character strings, called *patterns*, in another string, called a *text*. In this paper, we consider matching problem for patterns with *pictures*. For example, let A be a picture for a, b, ..., z, and N for 0, 1, ..., 9. Then we deal with patterns like abNN, a7$NNNNA$, ..., etc.

For string patterns, three matching algorithms are widely known: the Knuth-Morris-Pratt (KMP)[11], the Boyer-Moore (BM)[7], and the Aho-Corasick (AC)[1]. While the first two are for a single pattern, the third can simultaneously deal with multiple patterns. In this method, from a collection of patterns a finite state machine is built which finds all occurrences of the patterns in a single pass through a text. Such a machine is called a *pattern matching machine*, pmm for short. It runs in linear time proportional to the text length. Moreover, the construction of the pmm takes only linear time proportional to the sum of the lengths of the patterns. Fig. 1

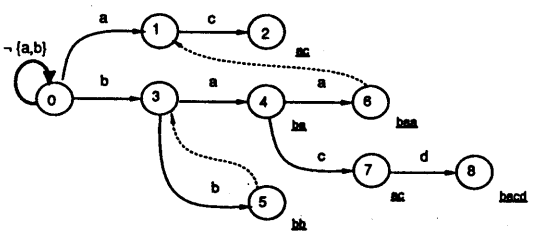


Fig. 1: The pmm for {ac, ba, bb, baa, bacd}

shows the pmm for patterns ac, ba, bb, baa, bacd. The solid arrows represent the *goto* function, and the broken arrows represent the *failure* function, where broken arrows to the state 0 from all but the states 0, 5 and 6 are omitted. The underlined strings below the states mean the *output* function.

Consider how to construct a pmm for multiple patterns with pictures. It is easy when the patterns consist only of pictures. We can construct a pmm easily if we take each picture as a character. However, it is difficult for patterns with both characters and pictures.

In this paper, we present an algorithm that builds an efficient pmm for multiple patterns with pictures. The machine can be built easily and quickly. While the number of states is not always minimum, it is reasonably decreased. When all the pattern are strings, the algorithm produces the same machine as the AC algorithm. Moreover, we can easily transform it into a deterministic finite state machine in the same manner as Aho-Corasick's [1].

This paper is based in large part on a report [18].

2 Matching problem for patterns with pictures

Let Σ be a finite alphabet and let Σ^* be the the free monoid generated by Σ. We call an element w of Σ^* a *string* and the length of w is denoted by $|w|$. Let $\Sigma^+ = \Sigma^* - \{\varepsilon\}$ where ε is the *empty string*. We say that u is a *prefix* and v is a *suffix* of uv, where u, v are strings. We shall denote by $PRE(u)$ the set of prefixes of a string u, and by $SUF(u)$ the set of suffixes of u. For strings u, w, we say that u *occurs at position* i *in* w iff there exist strings x, y such that $w = xuy$ and $i = |x| + 1$.

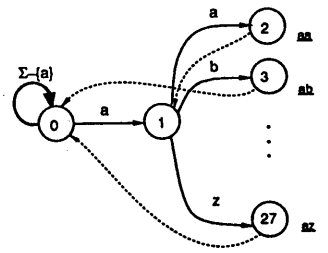
Let $\Delta = \{A_1, \ldots, A_p\}$ be a collection of disjoint nonempty subsets of Σ, i.e., $\emptyset \neq A_i \subseteq \Sigma$ and $A_i \cap A_j = \emptyset$ $(i \neq j)$. An element of Δ is called a *picture*. A *pattern* is chosen from $(\Sigma \cup \Delta)^+$, and a *text* from Σ^*. For a pattern π and a string w, we say that π *occurs at position* i *in* w iff there exists a string $u \in \pi$ such that u occurs at position i in w. Then, we consider the following problem:

> Given a collection of patterns $\Gamma = \{\pi_1, \ldots, \pi_k\}$ and a text T, to find all positions at which π_i occurs in T for $i = 1, \ldots, k$.

Now, let us consider to solve this problem by using the AC type pmm. From here on, we assume that

$$A = \{\mathrm{a}, \mathrm{b}, \ldots, \mathrm{z}\}, \quad N = \{0, 1, \ldots, 9\},$$
$$\Sigma = A \cup N, \qquad \Delta = \{A, N\}.$$

Method 1 The naive solution is to construct the pmm for all strings in patterns. That is, if the given pattern is aA, then we construct the pmm for strings aa,ab,...,az as below.

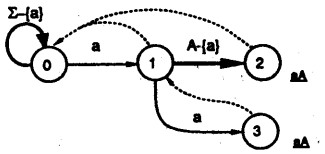


However, for a pattern A^m the number of states of the pmm is

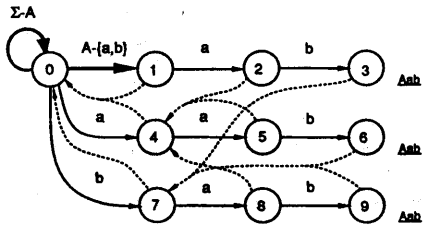
$$1 + 26 + 26^2 + \ldots + 26^m = \frac{26^m - 1}{25}.$$

This method thus increases the number of states very large.

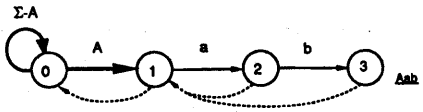
Method 2 Another solution is as follows: We determine the family of disjoint subdivided pictures from Δ and the characters occurring in the patterns, then we construct a pmm by taking each of these sub-pictures as a characters. For the pattern aA, we divide A into {a} and $A - \{\mathrm{a}\}$ to obtain:



However, for the pattern Aab, since A is divided into {a}, {b} and $A - \{a, b\}$, we obtain:



This pmm is not efficient, because the following pmm suffices for Aab:



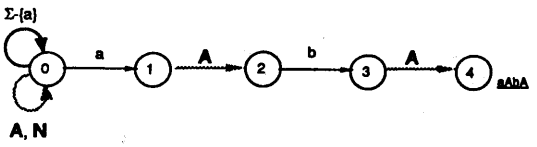
In this case, on the goto edge labeled A from 0, it is not necessary to distinguish each character in A, hence there is no need to make the edge branch off.

These observations tell us, for each goto edge labeled by a picture, to make it branch off to other states only when necessary. To do this, during the construction of the failure function, we shall make such edges branch off according to the values of the failure function. Next section describes our algorithm, based on this idea, for constructing an efficient pmm.

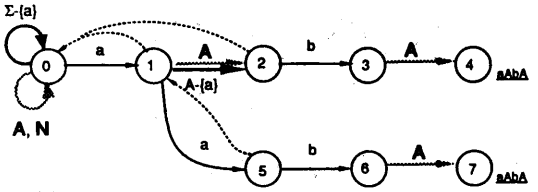
3 The algorithm

The algorithm for constructing the pmm consists of two parts, which are summarized in Algorithm 1 and 2. We illustrate their behaviors by the following examples.

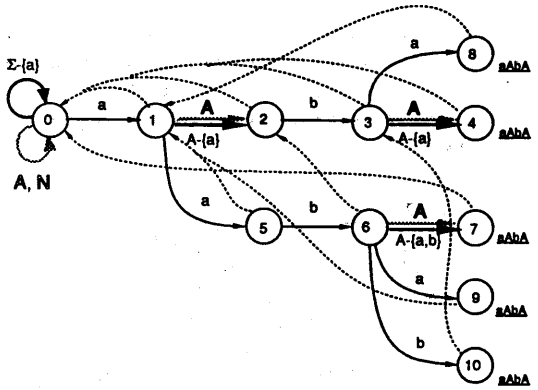
Example 1 Suppose aAbA is the pattern. In the first part, we treat each picture (denoted by outline letter) as a character, and obtain the graph:



In the second part, we construct the failure function and make goto paths branch off according to need. We first set $f(1) = 0$ since it is the state of depth 1. Then, we would compute the failure function for all states recursively, in nearly the same manner as the Aho-Corasick method. We would set $f(2) = 0$. However, if the input symbol on which we have made a goto transition from 1 to 2 is a, the failure value should be 1; Otherwise, 0. Therefore we shall make the edge branch off only for a to obtain:

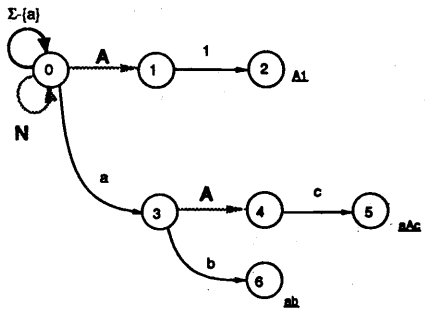


Note that we have copied the subtree whose root is 2 to the new state 5. Continuing in this fashion, we obtain:

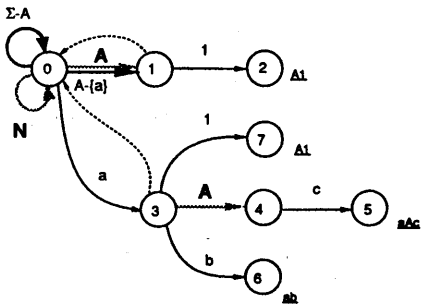


We next give an example for multiple patterns.

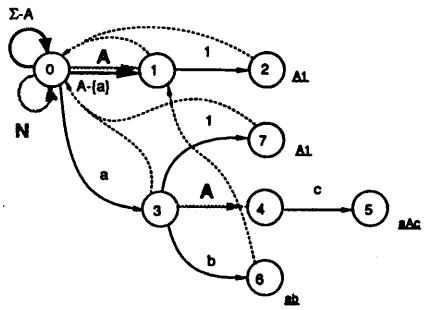
Example 2 Suppose that $A1$, aAc, ab are the patterns. In the first part, we obtain the graph:



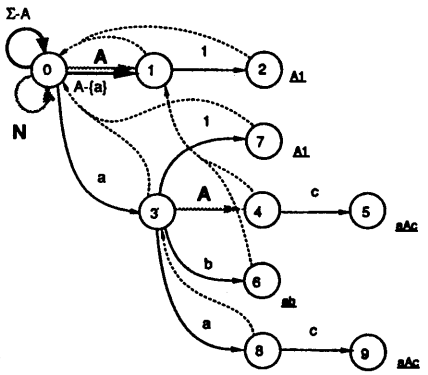
In the second part, we initially set the graph as below:



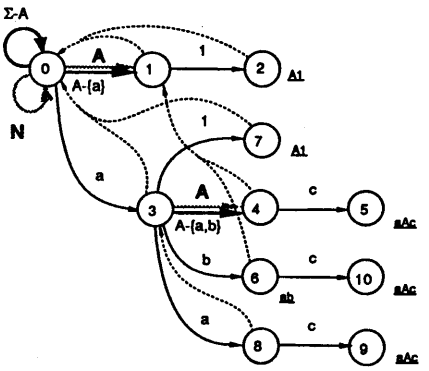
Note that the subtree whose root is 1 has been copied to the state 3. We then inspect the goto edge from 1 and get $f(2) = 0$. Now, we inspect the edges from 3. We first inspect every edge labeled by a character, and we get:



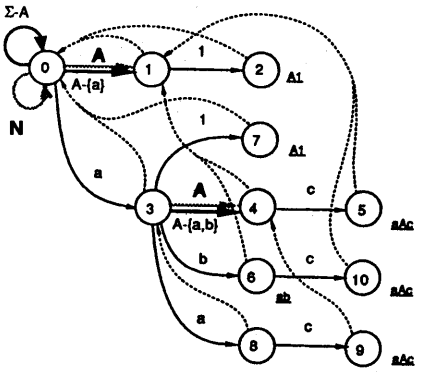
We next inspect every edges labeled by a picture, and we set $f(4) = 1$. Now we determine the next state from 3 for each character in the picture A. For the character a, the failure value should be 3, hence we make the edge branch off as below:



For b, since there already exists an edge labeled by b to the state 6, we shall copy there the subtree whose root is 4. For the other characters in A, the next states should be all 4. Thus we obtain:



Continuing in this fashion, we can complete the pmm as is shown below:



Thus, our algorithm simply produces an efficient pmm for multiple patterns with pictures. The text searching algorithm, which is summarized in Algorithm 3, is exactly the same as Aho-Corasick's.

4 Validity of the Algorithm

A pmm M is said to be *valid* for a set of patterns Γ when with M Algorithm 3 indicates that pattern π ends at position i of text T iff there exists $y \in \pi$ with $T = uyv$ and $|uy| = i$. This section shows the pmm produced by Algorithm 1 and 2 is valid.

Remark 1 Let $b_1, b_2, \ldots, b_m \in \Sigma$. Then, the following are equivalent.

(1) There exists a pattern π such that for some $y \in \Sigma^*$, $b_1 b_2 \ldots b_m y \in \pi$.

(2) There exists a sequence of non-zero states $r_1, r_2, \ldots, r_m$ such that

$$g(0, b_1) = r_1, \quad g(r_i, b_{i+1}) = r_{i+1}\ (1 \leq i < m).$$

Then we define sets and a mapping as follows: For a given set of patterns $\Gamma = \{\pi_1, \ldots, \pi_k\}$, we put $K = \pi_1 \cup \ldots \cup \pi_k$. K is the set of all strings to be searched for. We shall denote by $PRE(u)$ the set of prefixes of a string u, and by $SUF(u)$ the set of suffixes of u. Put

$$W = \bigcup_{x \in K} PRE(x).$$

Define a mapping *state* from W into the set of non-negative integers by

$$\begin{cases} state(\varepsilon) = 0\,, \\ state(ua) = g(state(u), a) \quad (u \in W, a \in \Sigma, ua \in W)\,. \end{cases}$$

Note that *state* is *well-defined* because of Remark 1. We also put $Q = \{state(u) | u \in W\}$, which is the set of all states reachable from the initial state 0. For all $s \in Q$, we put $PATH(s) = \{u \in W | state(u) = s\}$. It should be noted that for any $s, t \in Q$, $s \neq t$ implies $PATH(s) \cap PATH(t) = \emptyset$.

We are now ready to characterize the goto, failure and output functions produced by Algorithm 1 and 2.

Lemma 1 Let $\pi = X_1 \dots X_m$ be a pattern. Then, for any j with $1 \leq j \leq m$, there uniquely exists a nonempty subset I of Q such that $X_1 \dots X_j = \bigcup_{s \in I} PATH(s)$ and $depth(s) = j$ for any $s \in I$, where $depth(s)$ denotes the depth of s, i.e., the length of the shortest goto path from 0 to s.

Proof. By induction on j. □

This lemma claims that the goto function g is valid in a simple sense. But we have to show that Algorithm 2 produces sufficient branchings of the goto paths to compute the valid failure function.

Lemma 2 Let $s \in Q$ with $s \neq 0$. Let u be a string in $PATH(s)$, and let v be the longest string in $(SUF(u) - \{u\}) \cap W$. Then, $v \in PATH(f(s))$.

Proof. By induction on the depth of s. □

Concerning with the output function out, the following lemma holds.

Lemma 3 For all $s \in Q$ with $s \neq 0$,

$$\begin{aligned} out(s) &= \{\pi \in \Gamma \mid PATH(s) \subseteq \Sigma^*\pi\} \\ &= \{\pi \in \Gamma \mid PATH(s) \cap \Sigma^*\pi \neq \emptyset\}. \end{aligned}$$

Proof. It suffices to prove the following:

(1) $out(s) \subseteq \{\pi \in \Gamma | PATH(s) \subseteq \Sigma^*\pi\}$.

(2) $\{\pi \in \Gamma | PATH(s) \cap \Sigma^*\pi \neq \emptyset\} \subseteq out(s)$.

We shall first prove (1) by induction on the depth of s. It follows from the construction of out and Lemma 1 that

$$out(s) = \{\pi \in \Gamma | PATH(s) \subseteq \pi\} \cup out(f(s)),$$

so it clearly suffices to show that $out(f(s)) \subseteq \{\pi \in \Gamma | PATH(s) \subseteq \Sigma^*\pi\}$. By the induction hypothesis,

$$out(f(s)) \subseteq \{\pi \in \Gamma | PATH(f(s)) \subseteq \Sigma^*\pi\}.$$

Since $PATH(s) \subseteq \Sigma^* PATH(f(s))$ by Lemma 2, if $PATH(f(s)) \subseteq \Sigma^*\pi$ then $PATH(s) \subseteq \Sigma^*\pi$. Thus we complete the proof of (1).

We shall then prove (2) by induction on the depth of s. Since

$$out(s) = \{\pi \in \Gamma | PATH(s) \cap \pi \neq \emptyset\} \cup out(f(s))$$

and by the induction hypothesis, it suffices to see that, for any $\pi \in \Gamma$, if $PATH(s) \cap \Sigma^*\pi \neq \emptyset$ then $PATH(s) \cap \pi \neq \emptyset$ or $PATH(f(s)) \cap \Sigma^*\pi \neq \emptyset$. Suppose that $PATH(s) \cap \Sigma^*\pi \neq \emptyset$. Let $u \in PATH(s) \cap \Sigma^*\pi$, and let $u = u'\alpha$ with $\alpha \in \pi$. If $u' = \varepsilon$, $u = \alpha \in PATH(s) \cap \pi$; Otherwise, take v for u as in Lemma 2. Then, since $\alpha \in (SUF(u) - \{u\}) \cap W$, α must be a suffix of v, hence $v \in PATH(f(s)) \cap \Sigma^*\pi$. Thus we complete the proof of (2). □

The following lemma characterizes the behavior of Algorithm 3 on a text $T = a_1 a_2 \dots a_n$.

Lemma 4 After jth pass through the for-loop, Algorithm 3 will be in state s iff $PATH(s)$ contains the longest string in $SUF(a_1 a_2 \dots a_j) \cap W$.

Proof. By induction on j. □

We now have the following theorem.

Theorem 1 The pmm M produced by Algorithm 1 and 2 is valid.

Proof. By Lemma 3 and 4. □

5 Time complexity

It is obvious that the text searching algorithm runs in linear time proportional to the text length. We then discuss the time complexity of the algorithm for constructing the pmm. Clearly, its first part takes only linear time proportional to the sum of the lengths of the patterns. The second part does so if the patterns consist only of characters, or only of pictures. However, it is not so simple when the patterns contain both characters and pictures. Our algorithm is designed to do branchings of the goto paths according to need during the construction of the failure function so as to decrease the number of states. The cost varies depending on how the paths will branch off. However, even in the worst case, it is bounded by those of Method 1 and 2 described in Section 2.

Consider the cost of Method 1. We denote by $\sharp(X)$ the number of elements in a set X. Then, for a pattern $\pi = X_1 X_2 \dots X_m$, the number of strings belonging to π is given by

$$\sharp(\pi) = \sharp(X_1) \cdot \sharp(X_2) \cdot \dots \cdot \sharp(X_m).$$

Note that $\sharp(\pi) = 1$ if π is a string pattern. We also denote by $|\pi|$ the length of a pattern π. Suppose that $\Gamma = \{\pi_1, \pi_2, \dots, \pi_k\}$ is the set of patterns. Then, Method 1 takes to construct the pmm linear time proportional to

$$\sum_{i=1}^{k} \sharp(\pi_i) \cdot |\pi_i|.$$

We then consider Method 2. Let c_i be the number of different characters which appear in the patterns at least once and which belong to the picture A_i, for each $i = 1, \dots, p$. Let $d(X) = 1$, if $X \in \Sigma$; $c_i + 1$, if $X = A_i$. Put $d(\pi) = d(X_1) \cdot d(X_2) \cdot \dots \cdot d(X_m)$, for a pattern

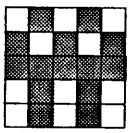
$\pi = X_1 X_2 \ldots X_m$. Then, the cost of Method 2 is linearly proportional to

$$\sum_{i=1}^{k} d(\pi_i) \cdot |\pi_i| .$$

Unless a large number of different characters appear in the patterns, $d(\pi_i)$ is much smaller than $\|(\pi_i)$. Moreover, if many characters appear in the patterns, accordingly the number of occurrences of pictures in the patterns is small, hence $d(\pi_i)$ will be 1 frequently.

6 An application to the two-dimensional pattern matching

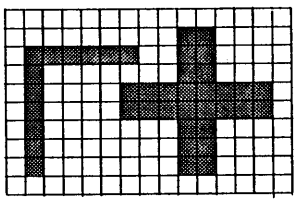
In the two-dimensional pattern matching problem, both pattern and text are two-dimensional arrays of characters. Bird [6] described an algorithm to solve this problem by using the AC method. Suppose that $\Sigma = \{\blacksquare, \square\}$, and let the pattern be



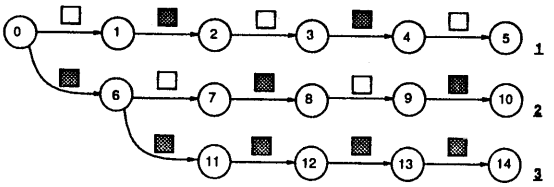
The method by Bird [6], regarding each row as a string pattern, builds a pmm as shown in Fig. 2 (a). Then, the pmm runs on the text row by row searching for the rows of the pattern array (*row-matching*). On the other hand, the machine shown in Fig. 2 (b) is used to determine whether or not the entire pattern array occurs in the text (*column-matching*). The algorithm takes $O(n_1 \cdot n_2)$ time to find all occurrences of the pattern in a text of size $n_1 \times n_2$.

However, this method has the following defect: Suppose that we would detect

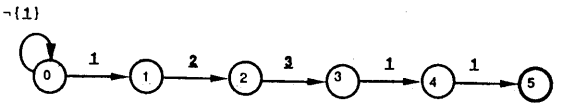
in the text



If we search for the rectangular pattern

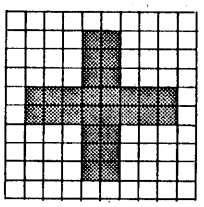


(a) row-matching



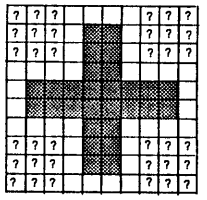
(b) column-matching

Fig. 2: The Bird method



by using the above method, we can not find it in the text.

Contrary to this, our method is able to deal with the pattern containing wild cards, i.e., pictures '?', such as



hence we can find the cross in the text.

7 Concluding remarks

We have presented an efficient matching algorithm for patterns with pictures. Since it is a natural extension of the AC algorithm, it has many possible applications.

Japanese texts consist of both 1-byte and 2-byte characters with shift codes. Shirohara and Arikawa [3] developed an algorithm to build a pmm for Japanese texts, based on the AC algorithm. It runs on a Japanese text without loosing the efficiency, taking each byte as an input symbol. If we combine our algorithm with this, we can deal with not only 1-byte pictures but also some 2-byte pictures, such as *kanji*, *hiragana*, etc., by treating them as concatenations of two 1-byte pictures.

When editing texts we often need replace some words by other words. Arikawa and Shiraishi [4] devised a multiple key replacement algorithm, which uses a generalized sequential machine produced in nearly the same manner as the AC machine. Our algorithm may improve the space efficiency of this method in some cases [17].

The pattern matching problem is simple, but very important for both theoretical and practical purposes. Various many studies on it have been done, some of which should be mentioned in the rest of the paper. We denote further by m, n the lengths of the pattern and the text, respectively.

The average case running time of the BM algorithm is said to be *sublinear*. Rivest [14] proved that no algorithm could solve the pattern matching problem in sublinear time even in the worst case. Yao [20] showed that the run time efficiency on the average of any matching algorithm could not be better than $O(n(\log m)/m)$, and this lower bound is achieved by Bailey and Dromey's algorithm [5].

The BM algorithm in the worst case takes $O(mn)$ time to lacate all occurrences of the pattern in the text. It can be modified so that the worst case running time is $O(n)$, not depending on m [9, 2, 15]. Guibas and Odlyzko [10] proved that the BM algorithm performs only at most $4n$ character comparisons when the pattern does not occur in the text .

The AC algorithm is an extension of the KMP algorithm to the multiple pattern problem. Similarly, several fast algorithms for multiple patterns have been devised as extensions of the BM algorithm [8, 12, 16].

Approximate pattern matching has also been studied. Three editing operations for a string are considered: insertion, deletion and substitution of a letter. The *distance* between two strings are defined as the minimum total number of such editing steps needed for converting one of the strings to the other. The problem is, for a given integer $k \geq 0$, to find all substrings of the text each of which has a distance of at most k from the pattern. Landau and Vishkin [13] presented an algorithm to solve the problem that runs in $O(m^2 + k^2 n)$ time. If only the substitution operation is allowed, it takes $O(k(m \log m + n))$ time. Ukkonen [19] proposed another interesting method: A deterministic finite automaton is built which accepts the set of all strings having a distance of at most k from the pattern. Although the construction of such automaton is not so efficient, this method is useful in some applications since the text scanning requires only $O(n)$ time.

References

[1] A. V. Aho and M. J. Corasick. Efficient string matching: An aid to bibliographic search. *Comm. ACM*, 18(6):333–340, June 1975.

[2] A. Apostolico and R. Giamcarlo. The Boyer-Moore-Galil string searching strategies revisited. *SIAM J. Comput.*, 15(1):98–105, February 1986.

[3] S. Arikawa and T. Shinohara. A run-time efficient realization of Aho-Corasick pattern matching machine. *New Generation Comput.*, 2(2):171–186, 1984.

[4] S. Arikawa and S. Shiraishi. Pattern matching machines for replacing several character strings. *Bulletin of Informatics and Cybernetics*, 21(1–2):101–111, 1984.

[5] T. A. Bailey and R. G. Dromey. Fast string searching by finding subkeys in subtext. *Inf. Process. Lett.*, 11(3):130–133, 1980.

[6] R. S. Bird. Two dimensional pattern matching. *Inf. Process. Lett.*, 6(5):168–170, October 1977.

[7] R .S. Boyer and J. S. Moore. A fast string searching algorithm. *Comm. ACM*, 20(10):762–772, October 1977.

[8] B. Commentz-Walter. A string matching algorithm fast on the average. In H. A. Maurer, editor, *Sixth International Colloquim on Automata, Languages and Programming*, pages 118–132. Springer-Verlag, 1979. Lecture Notes in Computer Science, 71.

[9] Z. Galil. On improving the worst case running time of the Boyer-Moore string matching algorithm. *Comm. ACM*, 22(9):505–508, September 1979.

[10] L. J. Guibas and A. M. Odlyzko. A new proof of the linearity of the Boyer-Moore string searching algorithm. *SIAM J. Comput.*, 9(4):672–682, November 1980.

[11] D. E. Knuth, J. H. Morris, and V. R. Pratt. Fast pattern matching in strings. *SIAM J. Comput.*, 6(2):323–350, June 1977.

[12] G. Kowalski and A. Meltzer. New multi-term high speed text search algorithms. In *The First International Conference on Computer and Applications*, pages 514–521, 1984.

[13] G. M. Landau and U. Vishkin. Efficient string matching in the presence of errors. In *Proc. of 26th Annual Symposium on Foundation of Computer Science*, pages 126–136, 1985.

[14] R. L. Rivest. On the worst-case behavior of string searching algorithm. *SIAM J. Comput.*, 6(4):669–674, 1977.

[15] I. Semba. An efficient string searching algorithm. *J. Inf. Process.*, 8(2):101–109, 1985.

[16] M. Takeda. A proof of the correctness of Uratani's string searching algorithm. Technical Report RIFIS-TR-CS-7, Res. Inst. of Fundamental Information Science, Kyushu University, October 1988.

[17] M. Takeda. Efficient multiple string repracing with pictures. Technical Report RIFIS-TR-CS-13, Res. Inst. of Fundamental Information Science, Kyushu University, March 1989.

[18] M. Takeda. A fast matching algorithm for patterns with pictures. Technical Report RIFIS-TR-CS-11, Res. Inst. of Fundamental Information Science, Kyushu University, March 1989.

[19] E. Ukkonen. Finding approximate patterns in strings. *Journal of Algorithms*, 6:132–137, 1985.

[20] A. C.-C. Yao. The complexity of pattern matching for a random string. *SIAM J. Comput.*, 8(3):368–387, August 1979.

Algorithm 1
input: A collection of patterns $\Gamma = \{\alpha_1, \ldots, \alpha_k\}$.
output: Partially computed functions g and out.
method:

```
begin
    nst := 0;
    for i := 1 to k do enter(α_i);
    for ∀X ∈ Δ ∪ Σ such that g(0, X) = fail do
        g(0, X) := 0
end


procedure enter(B_1 B_2 ... B_m):
begin
    state := 0;
    for j := 1 to m do
        if g(state, B_j) ≠ fail then
            state := g(state, B_j)
        else begin
            g(state, B_j) := nst;
            state := nst;
            nst := nst + 1
        end;
    out(state) := {B_1 B_2 ... B_m}
end
```

Algorithm 2
input: Partially computed functions g and out.
output: Functions g, f and out.
method:

```
begin
    queue := empty;
    for ∀c ∈ Σ such that g(0, c) = s ≠ 0 do begin
        queue := queue · s;
        f(s) := 0
    end;
    for ∀X ∈ Δ such that g(0, X) = s ≠ 0 do begin
        queue := queue · s;
        f(s) := 0;
        for ∀c ∈ X do
            if g(0, c) = t ≠ 0 then
                copy_subtree(s, t)
            else
                g(0, c) := s
    end;
    while queue ≠ empty do begin
        let queue = r · tail
        queue := tail;
        for ∀c ∈ Σ such that g(r, c) = s ≠ fail do begin
            queue := queue · s;
            fst := f(r);
            while g(fst, c) = fail do fst := f(fst);
            f(s) := g(fst, c);
            out(s) := out(s) ∪ out(f(s))
        end;
        for ∀X ∈ Δ such that g(r, X) = s ≠ fail do begin
            queue := queue · s ;
            fst := f(r);
            while g(fst, X) = fail do fst := f(fst);
            f(s) := g(fst, X);
            for ∀c ∈ X do
                if g(r, c) = t ≠ fail then
                    copy_subtree(s, t)
                else begin
                    fst := f(r);
                    while g(fst, c) = fail do fst := f(fst);
                    st := g(fst, c);
                    if st ≠ f(s) then begin
                        new := nst;
                        nst := nst + 1;
                        copy_subtree(s, new);
                        g(r, c) := new;
                        queue := queue · new;
                        f(new) := st;
                        out(new) := out(new) ∪ out(f(new))
                    end else g(r, c) := s
                end;
            out(s) := out(s) ∪ out(f(s))
        end
    end
end


procedure copy_subtree(st1, st2) :
begin
    queue2 := {(st1, st2)};
    while queue2 ≠ empty do begin
        let queue2 = (r1, r2) · tail
        queue2 := tail;
        out(r2) := out(r2) ∪ out(r1);
        for ∀X ∈ Δ ∪ Σ such that g(r1, X) = s ≠ fail do begin
            if g(r2, X) = fail then begin
                st := nst;
                nst := nst + 1;
                g(r2, X) := st
            end else st := g(r2, X);
            queue2 := queue2 · (s, st)
        end
    end
end
```

Algorithm 3
input: A text string $T = a_1 a_2 \ldots a_n$ and a pattern
matching machine M with functions g, f and out.
output: Locations at which patterns occur in T.
method:

```
begin
    state := 0;
    for q := 1 to n do
        begin
            while g(state, a_q) = fail do state := f(state);
            state := g(state, a_q);
            if out(state) ≠ empty then print q, out(state)
        end
end
```